Software Refactoring Process for Adaptive User-Interface Composition

Anthony Savidis, Constantine Stephanidis
Institute of Computer Science - FORTH and
Department of Computer Science - University of Crete, Crete, Greece
{as,cs}@ics.forth.gr

ABSTRACT
Adaptive user-interface composition is the ability of a software system to: (a) compose its user-interface at runtime according to a given deployment profile; and (b) to possibly drop running components and activate better alternatives in their place in response to deployment profile modifications. While adaptive behavior has gained interest for a wide range of software products and services, its support is very demanding requiring adoption of user-interface architectural patterns from the early software design stages. While previous research addressed the issue of engineering adaptive systems from scratch, there is an important methodological gap since *we lack processes to reform existing non-adaptive systems towards adaptive behavior*. We present a stepwise transformation process of user-interface software by incrementally upgrading relevant class structures towards adaptive composition by treating adaptive behavior as a cross-cutting concern. All our refactoring examples have emerged from real practice.

Author Keywords
Adaptive User Interfaces, Software Process, Software Engineering, Source Code Refactoring

ACM Classification Keywords
D.2.2 [*Design Tools and Techniques*]: User Interfaces; D.2.7 [*Distribution, Maintenance, and Enhancement*]: Restructuring, Reverse Engineering, and Reengineering

General Terms
Design, Human Factors

INTRODUCTION

Our work focuses on the ability of a software system to adaptively compose (or decompose) its user-interface during runtime according to deployment parameter values, the latter being user and context related information. We put emphasis on runtime component selection, composition and replacement from a repository of alternative user-interface component implementations.

EICS'10, June 19–23, 2010, Berlin, Germany.

Contribution We introduce a *user-interface refactoring process* to accommodate adaptive behavior into existing, even non-adaptive, user-interface implementations. Refactoring [14], a term from mainstream software engineering, concerns the process of gradually applying small-scale source-level transformations [12] aiming to enhance a system's software design, however, *without affecting its domain-specific functionality*. Refactoring is an incremental design improvement activity rather than a software reengineering process. Thus, refactoring is seen as a stepwise code evolution process to enhance an existing design, rather than to radically reform the software design itself. Radical reformations are handled by reengineering processes, being more resource demanding as they introduce severe architectural modifications.

Impact Our approach, being a refactoring process, does not impose architectural refinements, meaning *it respects the original user-interface software architecture*. Instead, it emphasizes the introduction of targeted source code amendments, transforming gradually the class-level software design of user-interface components towards adaptive composition. The latter is accomplished because all transformations we adopt are existing refactoring patterns. Since refactoring is architecture-preserving refinement [14], so is our process. Technically, *we treat adaptive user-interface composition and updating as a cross-cutting concern* – sort of an *aspect* [11] - meaning it intersects with the software system design at specific points, however, without affecting the global architectural picture. When making a system adaptive, interaction redesign is unavoidable, while the evolving *interaction design space* can be managed by methods such as [18]. Our work focuses on the *interaction implementation space*, showing that adaptive composition is feasible without redesign, through stepwise refactoring transformations. In software engineering, redesign relates to macroscopic structural and architectural changes. Refactoring, due to its locally bound implications, *is not* considered a redesign activity.

Benefits Overall, our proposition is manifold: (a) we introduce a process to reform an existing non-adaptive interactive system towards adaptive behavior; and (b) we

accomplish the latter not via reengineering but through refactoring, meaning through our suggested refinements the original system architecture is preserved .

Scope Our method is applicable to interactive applications *implemented in OOP programming languages, whether imperative or functional*. The latter include C++, Java, C#, Visual Basic, Objective-C, Smalltalk, Python, Ruby, Perl, JavaScript, Lua, ActionScript, Self, Haskell, OCaml, Scheme and Common Lisp, in which the vast majority of user-interface software is currently written. Although we use C++ for our examples, those are simple and generic making porting to another language rather trivial.

ADAPTIVE COMPOSITION

We brief the notion of adaptive user-interface composition, using as an example an adaptive web browser [10]. The browser, relying on individual user profiles, offers to end-users a best-fit user-interface. The latter implies that:

(i) The user-interface design encompasses alternative versions of specific interactive components, each matching particular characteristics of the user profile;

(ii) The user-interface source code encompasses all such alternatives as fully implemented distinct user-interface components; and

(iii) The user-interface system regulates the adaptation-oriented runtime selection activation of such components, in a way optimally matching the individual end-user profile.

Adaptive composition is strongly linked with the notion of alternative versions of user-interface components, each such alternative matching particular instantiations of the deployment profile. Reflecting this concept, Figure 1 outlines part of the hierarchical component decomposition of the adaptive web-browser user-interface.

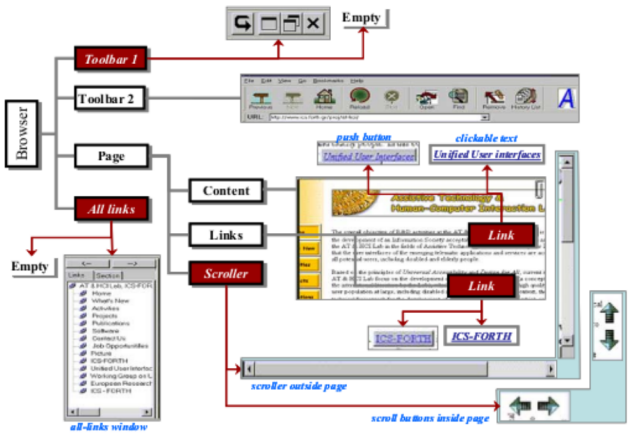

Figure 1: Excerpt from the user-interface component structure of the adaptive web browser; shaded (red) rectangles are adaptive components with alternative versions (following arrows).

Shaded rectangles are *adaptive components*, i.e. user-interface components for which various alternative component versions exist. Such alternatives are usually mutually-exclusive, meaning during runtime only one of them should be chosen and instantiated. For example, following Figure 1, for different end-users, links as either push-buttons or clickable underlined text may be chosen. In this browser, such decisions are taken during system startup, by a separate rule-based decision component [16].

The latter, for each adaptive component, decides the most appropriate component version to activate given the end-user and context-of-use profiles. It responds to requests of the form <***component A?***> via <***activate Aj***> messages, meaning for the adaptive component ***A*** the ***Aj*** *component* version is actually selected. Naming of user-interface components is arbitrary, while decision dispatching is handled within user-interface source code during runtime.

As seen, an inherent effect of adaptive behavior is that some user-interface components may have to be delivered with alternative versions, coexisting as part of the implementation, but with mutually-exclusive runtime presence. Such components are characterized as ***polymorphic***, referring as ***unimorphic*** to the rest [18]. An important property is revealed when exploring the effect of letting user-interface containers become *polymorphic*, i.e., having alternative containment structures. As shown in Figure 2, the entire user-interface hierarchy is now turned to polymorphic.

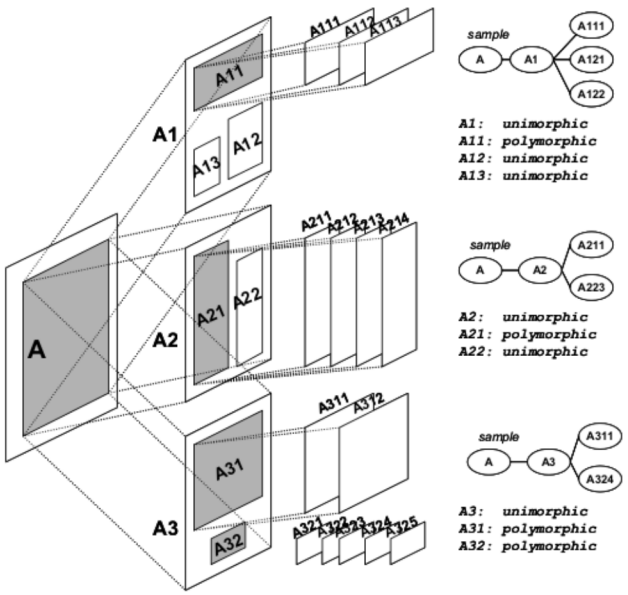

Figure 2: The concept of polymorphic user-interface containment due to adaptively polymorphic containers.

The latter is actually the most demanding aspect that has to be accommodated. It requires prescribe the transformation from non-adaptive implementations with traditional unimorphic user-interface component hierarchies, to adaptive ones, supporting polymorphic containment with dynamic adaptive activation. Thus, we consider this software requirement as the *governing feature* of our source code transformation process towards adaptive behavior.

Next we continue with an account of related work from the domains of adaptive systems, adaptive composition and software refactoring practices.

RELATED WORK

In software development, there are numerous categories of refactoring activities ranging from low-level local transformations like *Variable Renaming*, *Temp to Query*, *Method Extraction*, *Class Extraction* [7] and *Slice Extraction* [6], to higher-level wider modifications like *Extract Superclass* and *Aggregation Refactoring* [9].

Refactorings are also distinguished among primitive ones, being elementary and irreducible, to composite ones, prescribing sequences of primitive refactorings [13]. Overall, there is a large repertoire of generic transformations, all well documented and guaranteed to improve source code quality. All refactorings we adopt to infuse adaptive behavior are such well-known quality improving transformations.

Regarding adaptive user-interface behavior there are various propositions all of which suggest customized architectures or processes for a *development from scratch* approach. For example, the proposal of a *plasticity reference framework* [2] supports adaptation (retargeting) to different platforms. A model-driven approach for plasticity is proposed in [5]. The adoption of a task-driven user-interface architecture for adaptive ambient intelligence is proposed in [3], while adaptive information retrieval is addressed in [22] as a way of adaptive content composition. As with the previous propositions, our earlier work on user-interface adaptation [18], with the notion of unified user-interfaces, also required early adoption of our suggested user-interface architecture. All previous methods tend to be impractical for updating existing software systems to accommodate adaptive behavior for two primary reasons:

- The adaptive behavior is imposed as the dominant view of the user-interface architecture, although the domain-specific non interactive source code may be orders of magnitude larger compared to adaptivity-specific code
- They are quite diverse and their combined adoption in a single system is not investigated and might introduce practical issues due to differing architectural styles implied by varying goals: user intention extraction, dialogue automation, content adaptation, context adaptation, and cross-platform delivery

Naturally, propositions that do not address user-interface engineering but focus on the reasoning to decide adaptation, such as [15], may be adopted once they do not require architectural refinements.

In the context of our work we investigated the underlying software engineering disciplines to support adaptive behavior, seeking for a common denominator amongst the alternative approaches. It quickly turned out that, irrespective of the eventual adaptive behavior, all methods entail three fundamental concepts:

- ***user-interface component alternatives***
- ***rationale runtime component selection***
- ***adaptive activation or replacement***

Dynamic activation and replacement were very early recognized as the fundamental system actions to realize adaptive behavior [4]. Thus, once supported, virtually any designed adaptation scenario is implementable. The complexity, size and type of components widely vary, ranging from widgets to comprehensive dialogues, while adaptive activation may imply standalone presence, composition (aggregation) and replacement (substitution). In fact, these disciplines proved to be so fundamental that we could directly generalize from adaptive user-interfaces to adaptive software in general [17]. However, even in this general proposition the practicality issue was not resolved: the original system architecture had to be always refined since a software reengineering process was suggested. At this point, our proposal for a refactoring process addresses this issue, showing that we can effectively enable adaptive behavior by treating adaptive composition as a cross-cutting concern that can be accommodated with well-defined incremental source-code transformations. We continue with an elaboration of this refactoring process.

PROCESS DESCRIPTION

We adopt ***role*** as a responsibility-based notion for user-interface components, essentially abstracting over user-interface operations, and ***requirements*** to denote functional requirements specific to roles. The latter reflects recent trends in software-design [23] where the emphasis is shifted from functionality-driven class-based design to responsibility-driven role-based design. We can have alternative implementations for a given role, role implementations being actual components, with ***r(α)*** denoting the role of user-interface component ***α***. Following this we define how relationships among components emerge by respective relationships among their roles:

component ***a***	$\Leftrightarrow$	***a*** *implements* ***r(a)***
adaptive *component* ***a***	$\Leftrightarrow$	***a*** *implements* ***adaptive r(a)***
α ***contains*** β	$\Rightarrow$	*r(α)* ***contains*** *r(β)*
α ***deploys*** β	$\Rightarrow$	*r(α)* ***deploys*** *r(β)*
α ***indifferent***	$\Rightarrow$	*r(α)* ***indifferent*** *r(β)*

It should be noted that ***contains*** and ***deploys*** relate to aggregation and deployment at the functional level, not the user-interface layout. The previous definitions state that component relationships are implied by the relationships of their abstract operations or roles. Implementation wise, *roles map to components, and components map to concrete classes, modules or packages*. We will rely on these two fundamental software relationships to drive structural transformations, using mainly the refactoring catalogue of [7], for all adaptively contained or deployed components. Our refactoring process is outlined under Figure 3, prescribing: (a) three preparatory activities to extrapolate,

model, and represent information from data already available at the end of the user-interface (re)design phase; and (b) five concrete source code transformation activities to actually implement the adaptive behavior. We continue with the process details. All presented examples are taken from the implementation of the adaptive web browser [10].

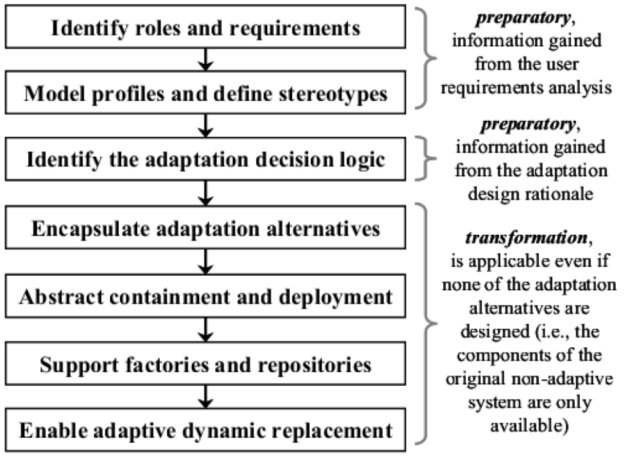

Figure 3: Adaptation-oriented user-interface refactoring process.

Identify roles and requirements

As mentioned earlier, adaptive composition involves at runtime the adaptation-driven selection and activation of dialogue components, from a pool of related alternatives, with the design aim to optimally support respective user tasks. In order to refactor a system for adaptivity, we should initially associate concrete roles to all implemented user-interface components, and identify their respective functional requirements. As an example, consider the roles and requirements defined below for the adaptive browser.

Roles	***Requirements***
Link	Allows activate a target page.
Scroller	Allows view parts of large pages.
AllLinks	Allows view all page links.
LinkTargeting	Allows target to a specific link.

What the previous list states is that *Link* is a user-interface role whose primary functional requirement is to facilitate activation of a target page. In case such brief information is not already part of the user-interface documentation, it can be easily extrapolated by reviewing the user-interface design outcomes or by referring to the designers.

Model profiles and define stereotypes

Supporting adaptation means delivering appropriate variations of system behaviour according to different deployment profiles. For interactive systems, such profiles concern user and usage contexts. Practically, the user-interface system exposes different user-interface profiles in response to deployment profiles. We use the term ***interface profile model*** to denote the model expressing the domain of variations of user-interface behaviour. Such a model enumerates the viable alternatives for adaptive roles. For example, the following model relates to the adaptive browser (a specification pseudo language is used):

```
interface profile model {
   Link          : { NoCancellation, WithCancellation }
   Scroller      : { OutsidePage, InsidePage }
   AllLinks      : { Supported, Unsupported }
   LinkTargeting : { Manual, Assisted }
}
```

The previous model designates the number of different adaptive composition possibilities in the browser, since every interface profile instance essentially represents a separate user-interface setup. Additionally, besides concrete roles like *Links* and *Scroller*, cross-cutting user-interface features, like *Language*, *Audio Feedback*, etc., can be well expressed. Using this model we can identify distinct setups as appropriate for sets of deployment profiles. For instance, let's consider the following instantiations of our simple interface profile model, called stereotypes:

```
interface stereotype ForNaiveUsers {
   Link          = WithCancellation
   Scroller      = OutsidePage
   AllLinks      = Unsupported
   LinkTargeting = Assisted
}

interface stereotype ForExpertUsers {
   Link          = NoCancellation
   Scroller      = InsidePage
   AllLinks      = Supported
   LinkTargeting = Manual
}
```

Intuitively, there is a rationale link between the chosen stereotype names, like *ForNaiveUsers*, and the respective values given to the fields of the interface profile model. Such interface profile instances are called *user-interface stereotypes* and they document in a readable way key scenarios of adaptive setup, making more explicit the deployment profile accommodated with such setups. For example, one anticipates that the *ForNaiveUsers* stereotype is normally targeted to user profiles implying a 'naïve end-user'. This step of distinguishing specific user-interface setups is optional. Its objective is to provide an initial understanding regarding the adaptivity potential of the original or refined user-interface implementation.

The next step is to define the *deployment profile model* representing information about the end-user and the usage context. User information may be stored in profile databases, may be gained from servers via unique user identifiers, can be extracted from a smart card, may be required user-input in a startup interaction session, or can be inferred at runtime from interaction monitoring and analysis. Similarly, context information, like location, environment noise, network

bandwidth and machine setup, may be provided using special-purpose equipment like sensors (for changing features) and a system-level profile (a registry for static features). Clearly, the definition of the deployment profile model should reflect information about the actual user population and the real environments of use of the software system. The following is an excerpt of the user profile model for the adaptive browser.

```
user profile model {
   ComputerLiteracy      : { Good, Average, Some, None }
   WebUse                : { Frequent, Casual, None }
   UserAge               : { 4 .. 90 }
   NativeLanguage        : Language
   LanguagesSpoken       : list of Language
}
user stereotype Naïve {
   ComputerLiteracy in { Some,  None}   or
   WebUse           in {Casual, None}
}
```

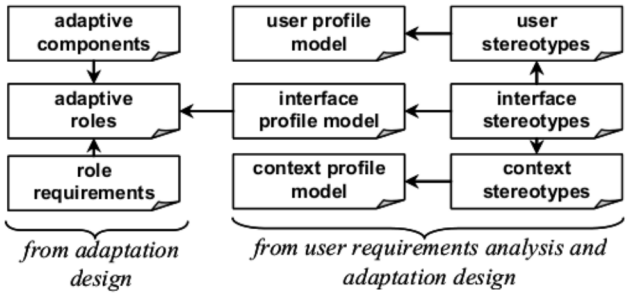

Figure 4: Defining roles, models and profiles - *first two phases*.

At this point, our two preparatory phases result in the information outlined under Figure 4, providing a clear view of the adaptation design goals, artifacts and requirements of the refactored interactive system.

Identify the adaptation decision logic

Typically, during an adaptation-oriented (re)design phase, the concrete design rationale of every adaptive user-interface component is defined and documented. Such rationale encompasses the actual conditions regulating the delivery and presence of an adaptive component during interaction (runtime). In other words, it defines *when* adaptive activation of components should be performed, and *which* of the available alternatives of adaptive components should be chosen. Only once such information is available the design phase gracefully concludes.

Conceptually, such rationale constitutes a form of decision logic for adaptive component activation or deactivation of user-interface components. Additionally, such logic rationally links the deployment profile with designed user-interface components, meaning it is user and context dependent. Linking with our previous notions, ***the adaptation decision logic links directly deployment profile attributes to interface profile attributes***. As a trivial example, the end-user native language is typically used to choose the user-interface language. Examples follow of adaptation rules expressed in a publicly available[1] decision language [16] supporting declarative rules with an imperative syntax, notice the use of stereotypes in decision conditions (*Link* being an adaptive component).

```
component Link {
        if  Naïve then   activate WithCancellation
        else             activate NoCancellation
}
component LinkTargeting {
        if  Naïve or Elderly then activate Assisted
        else                      activate Manual
}
```

Practically, developers may collect and implement the decision logic using any convenient method, including hard-coding in the user-interface implementation language, while express all profiles directly in XML. The most common approach is to introduce a separate special-purpose class responsible for decision making during runtime. Later, once the entire adaptation logic is consolidated and implemented, and the system is transformed to an adaptive one, alternative implementation techniques may be explored better fitting the notion of a decision-making module (like rule-based systems or logic programming methods). Due to the simplicity of this implementation task, and because it does not affect the existing user-interface system, we consider it more of a preparatory action rather than a transformation step.

Summing up, the job of this phase is the collection and formulation of the adaptation design rationale in a more formal style, being closer to a logic or algorithm form, i.e. a computable representation. Clearly, this preparatory phase reflects an intention to make interactive systems capable to execute adaptation-related logic so as to realize a runtime decision making for required adaptation actions. The outcome of this phase is outline in Figure 5, showing that it involves on outcomes of the first two phases.

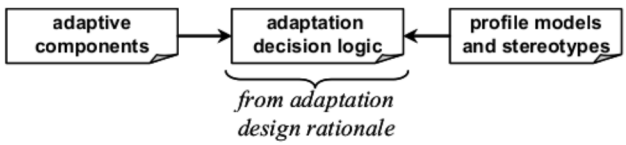

Figure 5: Identification of adaptation decision logic - *third phase*.

Encapsulate adaptation alternatives

During interaction, for every adaptive component, only its contextually appropriate version is always activated. Once an adaptation alternative is selected and activated, it should be indistinguishable to the rest of the user-interface system from its alternatives, thus ***related adaptation alternatives should comply with a common programming interface***,

[1] http://www.ics.forth.gr/hci/files/plang/DMSL.ZIP

like a common superclass. Since related components are linked by *deployment* or *containment*, we should prepare the ground for both cases through *Extract Hierarchy* and *Create Abstract Super Class* refactoring actions [7]. The example below depicts the simple transformations to support alternative types of links in the adaptive browser.

```
class Link {              ← common API of Link alternatives
    virtual void OnSelect (void)
                {…}       ←actions to get page for URL
    Link (const string& url);
};
class LinkAsLabel : public Link, public Label {
    virtual void OnMouseClick (void)
                { OnSelect(); }
};
class LinkAsButton: public Link, public Button {
    virtual void OnPress (void)
                { OnSelect(); }
};
```

In the previous code, the *Link* superclass concerns the adaptive *Link* component, while its subclasses correspond to its adaptation alternatives. As shown, the *Link* superclass encompasses non-interactive link functionality, storing the target URL and posting requests to retrieve the web document. Subclasses, like *LinkAsLabel* or *LinkAsButton*, need encompass within their interaction (event) handlers *OnSelect* invocations. This transformation is only a first step to make adaptation alternatives *look alike* in terms of software handling, and is outlined in Figure 6.

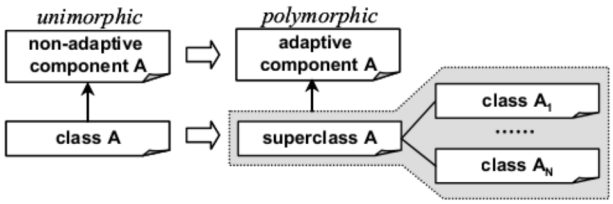

Figure 6: Class transformations to turn non-adaptive (unimorphic) components to adaptive (polymorphic) - *fourth phase*.

It is possible that user-interface components are not originally implemented as classes (object-oriented code), but as typical procedural code. In this case, we may turn them to real objects via *Convert Procedural Design to Objects* and *Extract Class* refactoring actions [7].

Abstract containment and deployment

Following the previous source code transformation, all user-interface components which are adaptation alternatives should comply with abstract programming interfaces. Now, we proceed to handling all cases of adaptive *deployment* and *containment*. For this purpose, we enforce the exclusive use of superclass references for all *contained* and *deployed* adaptive components. The gains from this rule are twofold: (i) reduced dependencies, by decoupling clients of adaptive components from the adaptation alternatives; and (ii) flexibility and extensibility, as one may drop, change or add adaptation alternatives without affecting respective clients.

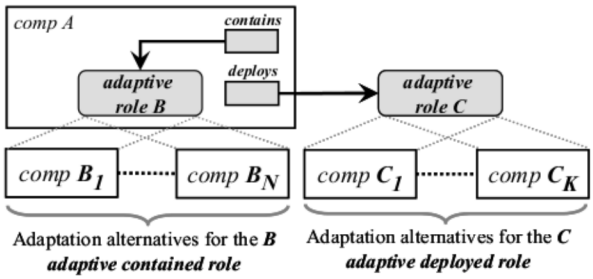

Figure 7: The notion of adaptive polymorphic containment and deployment for clients of adaptive components.

The notion of adaptive containment and deployment is illustrated under Figure 7. Clearly, without our rule, the software design becomes excessively complicated by turning clients dependent to every alternative version of their deployed and contained adaptive components. Following Figure 7, given the client component *A*, with *B* and *C* being two of its contained and deployed adaptive components, the following software design rules must hold:

- *B* is a superclass of $B_1,\ldots,B_N$
- *C* is a superclass of $C_1,\ldots,C_K$
- In *A*, only *B* and *C* references are allowed, not B_i or C_j

At a first glance it seems that adaptive containment and deployment are well handled via class abstraction. However, at the implementation level, there are fundamental differences amongst these two relationships:

- In *deployment*, the client commonly accepts a superclass reference to an ***externally created*** deployed component
- In *containment*, the container is responsible to ***internally create*** an instance of the contained component

In the deployment case, class abstraction suffices. However, in the containment case, the issue is how container clients can construct instances of contained adaptive components without actually referring to their subclass. We address this problem with a variation of the *Factory Design Pattern* [8], supporting name-based instantiation decoupled of subclasses. The later provides an infrastructure that can play the role of a *user-interface component repository*.

Support factories and repositories

In this step, besides container-contained decoupling, we also seek for an optimal decoupling of the decision rules from the rest of the source code. Practically, decision rules should refer to chosen component alternatives only by their design names, independently of package, module, class or instance names. Optimally, we should be able to directly adopt the user-interface component identifiers from the *interface profile model*. Additionally, there is no need for globally unique component identifiers, since, given an adaptive component, we need only use names that allow disambiguate among its specific adaptation alternatives. Technically, name-based component instantiation means adaptive components can dispatch instantiation requests issued by the decision engine.

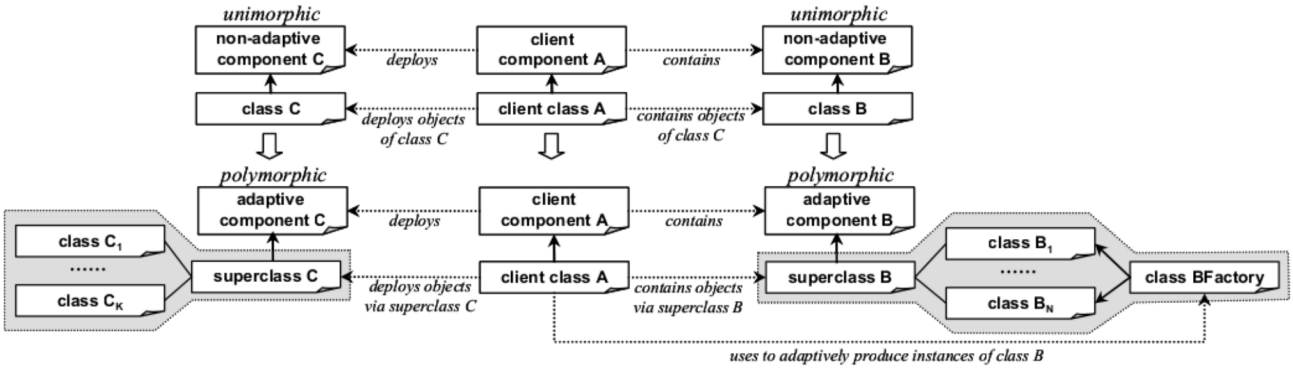

Figure 8: Outline of class transformations when applying abstract containment and deployment for adaptive components (fifth refactoring phase), and enabling adaptive instantiation of adaptation alternatives through factories (sixth phase).

The latter is implemented easily through a variation of the *Factory Design Pattern* [8] through a common transformation activity known as *Refactoring To Patterns* [10]. An example of applying this sort of transformation in the adaptive browser is provided under Figure 9.

```
The container component class (selected members only shown)
class Page {
   string linkCompId, scrollerCompId;
   Method to request and record all adaptation decisions:
   void Adapt(void) {
     linkCompId     = Rules::Decide("Link");
     scrollerCompId = Rules::Decide("Scroller");…
   }
   Method for adaptive Link instantiation:
   Link* NewLink (void) const
    { LinkFactory::New(linkCompId,this); }
   Method for adaptive Scroller instantiation:
   Scroller* NewScroller (void) const
    { ScrollerFactory::New(scrollerCompId,this); }
   Method to create the entire visible web page:
   void Produce (void) {
        Invokes NewLink() to adaptively produce links
        Invokes NewScroller() to adaptively add scrolling
   }
};
```

Figure 9: Te Factory pattern for adaptive contained components.

The *Adapt()* method of every client is called at startup to post requests to the decision engine for all adaptive contained or deployed components, while recording the returned identifiers of the chosen adaptation alternatives.

Helper methods are defined per contained adaptive component, like *NewLink()* and *NewScroller()*. The latter perform adaptive instantiation using their respective factory, such as *LinkFactory::New* which uses the corresponding component identifier *linkCompId*.

Finally, in the *Produce()* method, the *Page* container creates instaces of *Link* subclass compliant to the supplied component identifier using *NewLink()*. The combination of decision making and factory realizes adaptive *Link* instantiation.

The combined effect when combining the two previous transformation phases is depicted under Figure 8. Because clients of adaptive components are entirely decoupled from the respective adaptation alternatives, future adaptation-driven extensions are possible without affecting clients. It should be noted that, although we have drawn *BFactory* to depend on the B_i classes, there are methods to eliminate event this dependency, such as using the runtime *Virtual Constructor Pattern* of [17].

Enable dynamic adaptive replacement

During runtime, the user-interface can monitor, detect and infer changes for context and user attributes, such as: (a) user location and environment noise (context); and (b) habits and intentions (user). As a result, the necessary actions should be taken by the user-interface to guarantee it continues optimally fit the modified deployment profile. The latter may imply actions like instantiating extra components, as adaptive *prompting* or *recommendations* [1], while in some cases radical updates may be mandated like *switching* components to activate better alternatives [3]. The later imply a form of adaptive component replacement, as illustrated in Figure 10.

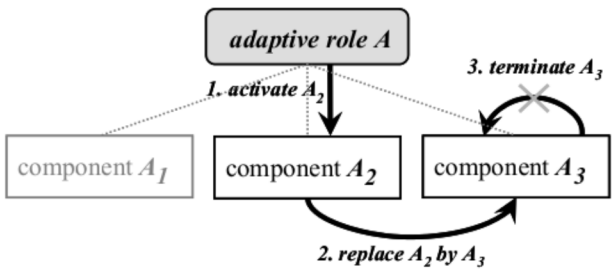

Figure 10: The notion of adaptive dynamic replacement.

To support adaptive component replacement we need terminate a component and activate its substitute, a special case being just dropping a component, i.e., replacing with an empty one. As variables referring to the replaced component may exist, we should update them to refer to the new component, nullification being also legal if components are dropped. A replaced component may possess state that

should be transferred to its substitute. Since all subclasses of adaptation alternatives should remain decoupled, we need support the necessary state copy functionality at the level of respective superclasses. Let's review a relevant refactoring for the adaptive browser using a *hypothetical* scenario for adaptive behaviour. Assume we monitor the use of the dropdown menu of the address bar to infer if the intent is visit sites whose address was explicitly typed in. Such sites are a subset of the addresses in the history list. Once we detect the intent, we remove the address bar, replacing it with a popup box having an address bar and a list of all typed addresses, annotated with some extra information. Firstly, we provide the decision logic for such adaptive replacement.

```
component Address {
   if user.Intent = VisitSitesWithTypedInAddresses then {
      if active(AddressBar) then
         cancel AddressBar
      activate AddressBox
   }
   else activate AddressBar
}
```

As shown, also support the case where the user intent was defined in the user-profile, not only inferred at runtime. Thus, the *AddressBox* may be the initial choice upon adaptive composition. But if the *AddressBar* was already chosen, then it is closed by putting *AddressBox* in its place. Such a sequence of *cancelations* followed by *activations* for an adaptive component is interpreted as replacement.

State copy and state transfer For the implementation of our scenario, we design the superclass of the *Address* adaptive component encompass the necessary state data - the address list - as shown below (access qualifiers and various data members are skipped).

```
class Address {
    Data type for transferable state upon replacement:
    class State {
        void Add (const string& address);
        State (const State&);
        State (const list<string>& _addresses);
    };
    Refinable (virtual) method to return a state copy:
    virtual const State GetState (void) const
        { return state; }
    Constructor through pre-initialised state:
    Address(const State& _st) : st(_st){…}
    State st;
};

class AddressBar : public Address {
    typedef Address::State State;
    AddressBar (const State& st) : Address(st){…}
};

class AddressBox : public Address {
    typedef Address::State State;
    AddressBox (const State& st) : Address(st) {…}
};
```

Construction of a new component using a copy of state is now supported by the *Address* superclass constructor, thus our problem of replacement by retaining (copying and transferring) state is solved.

Proxies for replaceable components We revisit the explicit reassignment of variables referring to the replaced component. Intuitively, it is suboptimal as it requires bookkeeping of all such variables. Instead, we need a way to have references to objects whose class can essentially change during runtime. The latter is prescribed by the *Proxy Design Pattern* [8], so once more we need refactor via *Refactoring to Patterns* [10], leading to the following source code transformation.

```
class AddressProxy : public Address {
   Address* addr;
   const State GetState(void) const
        { return addr->GetState(); }
   void Replace(Address* _addr) { addr = _addr; }
   Address* GetAddress(void)    { return addr; }
   AddressProxy (Address* a):   addr(a){}
};
```

Following the refactoring we introduce *AddressProxy* encapsulating an *Address* pointer. *For each replaceable component a single proxy instance is created, shared by all parties requiring access the component*. Following our example, the source code transformation to handle adaptive replacement of an *Address* component follows below, while their combined effect is depicted under Figure 11.

```
ProxyAddress* addr;        ←the shared proxy instance
addr->GetAddress()         ←how clients get the Address object
string addrCompId;         ←hold the Address component id

    Copying the state of the component under replacement
Address::State st(addr->GetState());

    Destroying the component under replacement
delete addr->GetAddress();

  Replacing with a new component initialised with the state copy
addr->Replace(Address::Factory(addrCompId, st));
```

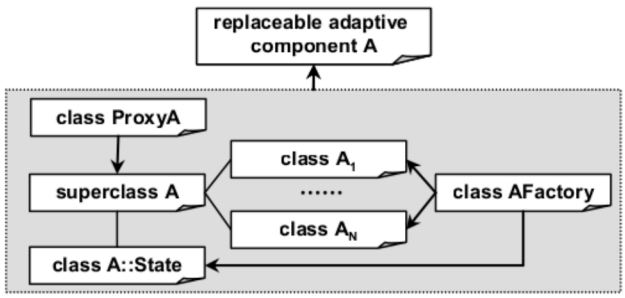

Figure 11: Transformations for state copy / transfer, and use of proxies for replaceable adaptive components (final phase).

DISCUSSION

Our method evolved from real-life within a decade, relating to five research projects, and more than 30 man years (two of them concerning [19] and [20]). The method was formulated iteratively through the following phases:

Phase I: Engineering for adaptivity

The first project concerned the development an adaptive web browser [20]. We started developing the core browser functionality and standard user-interface very early, far before our design process for identifying adaptation alternatives could come out with the first validated results. We deployed a software architecture designed to address adaptive composition, however, we lacked systematic ways to handle organization down to the source code level. Once we started encompassing alternatives we had to introduce abstraction and management layers, like factories, decorators, dispatchers, and proxies. At the end, it became evident that the software engineering activities to incorporate and handle adaptive functionality resulted in pretty standardized component implementation patterns. But such patterns were not reflected in our user-interface architecture. At that time we assumed this as to be natural, as architectures typically hide such low-level details.

Phase II: Extending for adaptivity

Following the previous phase, all software design techniques were formulated as an implementation cookbook for extending adaptive systems with adaptive functionality. Soon we started flirting with the idea that one may apply such software design techniques to extend any adaptive system, not only a system compliant with our user-interface adaptation architecture. But again we also felt this to be sort of 'wishful thinking', as we lacked methodologies to assess the effect of blending our software engineering method with other architectural propositions without actually having to fully build a system from scratch. So, we decided to put the idea 'into the refrigerator' until we had tangible evidence that it was either right or wrong.

Next, we started with the 2wear project[2] which concerned a toolkit [19] for adaptive, distributed, mobile and wearable user-interfaces, as part of the. We had to develop example adaptive applications using the tool, and document how third-party developers could extend our system with new user-interface components. Thus, we started documenting the extension library which involved software patterns like: abstraction, factories, proxies, and state sharing.

Since this was a sort of '*déjà vu*, we though to restart the idea that these patterns do apply to all adaptive systems. Now, the point is that our toolkit prescribed no particular user-interface architecture to developers: it enabled dynamically discover, grant, deploy and release user-interface services in a component-based fashion. Practically, it lets clients design and setup architectures, reflected in the way the components interoperate at runtime. This remark gave us the incentive that the lower-level software engineering patterns emerging when making an adaptive interactive system are truly universal, irrespective of the user-interface macro architecture. We observed that the software patterns consistently recurred for all user-interface components of our toolkit. The latter is a signal that adaptive behavior is a cross-cutting concern, applicable irrespective of the original architectural even if it was not designed to accommodate adaptive behavior.

Phase III: Refactoring for adaptivity

Eventually, we started examining ways to progressively transform any interactive system to support adaptive composition, without prerequisites or changes on its architecture. So, we needed incremental architecture-preserving transformations that could systematically modify a system towards adaptive behavior. The only category of architecture preserving source transformations is *software refactoring*. Hence, we carefully designed a process where all the necessary software patterns for adaptive composition are introduced through a series of existing refactorings. This way, we ended up with a refactoring process capable to incrementally transform an interactive system for adaptive user-interface composition.

Also, adaptive composition is a mechanism underpinning adaptation, with adaptation quality depending only on the interaction design process. Just like the quality of a given design is invariant, no matter the implementations, the engineering style for adaptive composition cannot interfere with adaptation quality. Thus, no such quality comparisons among reengineering and refactoring methods can apply.

SUMMARY AND CONCLUSIONS

We outlined a software refactoring process to support adaptive user-interface composition and replacement for systems not originally designed to support such adaptive behavior. Our work is motivated by the fact that, while adaptivity gains broad interest for software products and services, all known propositions imply development from scratch and adoption of architectural styles that may not necessarily interoperate with the domain-specific software architecture. Embodying adaptive behavior into existing systems can be only heuristically applied since to our knowledge we lack relevant transformation processes. Our approach reflects three key principles: (i) modeling of adaptive behavior as adaptive composition activities; (ii) treating composition and replacement as cross-cutting concerns with local design implications; and (iii) gradually transforming components through well-known existing refactoring activities.

Our process prescribes: (a) three preparatory activities, not directly affecting the source code, but aiming to reveal and model the primary system concepts cross-cutting with adaptive behavior; and (b) four specific transformation activities bringing eventually the user-interface system to an implementation state supporting adaptive composition and replacement. Our focus on software refactoring rather than on software reengineering is fundamental. More specifically, via reengineering we need a process to fuse

[2] http://2wear.ics.forth.gr/ (full source of 80K lines included)

two parallel system designs and architectures together: the original domain-specific system design and architecture, and the one implied by the need for adaptive behavior. Not only we lack today such processes for interactive systems, but we lack software reengineering processes for software systems in general [12]. By adopting a refactoring process we have the extra key benefit that after every transformation activity the system is always in a fully working state [14]. Overall, we believe that analogous refactoring processes for different categories of demanding user-interface features may lead to their easier adoption in real production systems.

REFERENCES

1. Adomavicius, G., Tuzhilin, A. (2005). Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions. IEEE Transactions on Knowledge and Data Engineering. 17(6), pp 734–749
2. Calvary, G., Coutaz, J., Thevenin, D. (2001): A Unifying Reference Framework for the Development of Plastic User Interfaces. Proceedings of EHCI2001 conference, Springer LNCS 2254, pp 173-192
3. Clerckx, T., Vandervelpen, C., Luyten, K., Coninx, K. (2006). A task-driven user-interface architecture for ambient intelligent environments. In proceedings of the IUI 2008 conference, ACM, pp 309-311
4. Cockton, G. (1993). Spaces and Distances - Software Architecture and Abstraction and their Relation to Adaptation. In Adaptive User Interfaces - Principles and Practice, Elsevier Science, pp 79-108
5. Collignon, B, Vanderdonckt, J, Calvary, G (2008). Model-Driven Engineering of Multi-target Plastic User Interfaces. IEE ICAS'08, pp 7-14
6. Ettinger, R., Verbaere, M. (2004). Untangling: A Slice Extraction Refactoring. In proceedings of the AOSD'4 conference, ACM, pp 93-101.
7. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D. (2000). Refactoring: Improving the Design of Existing Code. Addison Wesley
8. Gamma, E., Helm, R., Johnson,R., Vlissides, J. (1995). Design Patterns: Elements of Re-Usable Object-Oriented Software. Addison – Wesley
9. Johnson, R., Opdyke, W. (1993). Refactoring and Aggregation. In proceedings of the *First International Symposium on Object Technologies for Advanced Software*, Springer LNCS Vol. 742, pp 264-278
10. Kerievsky, J. (2004). Refactoring to Patterns. Addison-Wesley
11. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. M., Irwin, J. (1997). Aspect-Oriented Programming. In Proceedings of the European Conference on Object-Oriented Programming, Springer LNCS vol.1241, pp 220–242
12. Mens, T. Tourwe, T. (2004). A survey of software refactoring. In IEEE Trans. on Software Engineering 30(2), February 2004), pp 126-139
13. O Cinneide, M., Nixon, P. (2000). Composite refactorings for Java programs. Tech. Rep., Department of Computer Science, University College Dublin
14. Opdyke, W. (1992). Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks, Ph.D. thesis, University of Illinois at Urbana-Champaign
15. Paternò, F., Santoro, C., Scorcia, A. (2008). Automatically adapting web sites for mobile access through logical descriptions and dynamic analysis of interaction resources. ACM AVI 2008, pp 260-267
16. Savidis, A., Antona, M., Stephanidis, C. (2005). A Decision-Making Specification Language for Verifiable User-Interface Adaptation Logic. International Journal of Software Engineering and Knowledge Engineering. Vol. 15, No. 6 (December 2005), WSP, pp 1063-1095
17. Savidis, A. (2004). Dynamic Software Assembly for Automatic Deployment-Oriented Adaptation. In Elsevier Electronic Notes on Theoretical Computer Science (ENTCS), Vol. 127, Issue 3, pp 207-217
18. Savidis, A., Stephanidis, C. (2006). Unified User Interface Development: New Challenges and Opportunities. In J. Jacko & A. Sears (Eds.), The Human-Computer Interaction Handbook (2nd Edition). Mahwah, New Jersey: LEA, pp 1083-1106
19. Savidis, A., Stephanidis, C. (2005). Distributed interface bits: dynamic dialogue composition from ambient computing resources. ACM-Springer, Personal and Ubiquitous Computing 9(3), pp 142-168
20. Stephanidis, C., Paramythis, A., Sfyrakis, M., Savidis, A. (2001). A Case Study in Unified user-interface Development: The AVANTI Web Browser. In User-interfaces for All, Lawrence Erlbaum, pp 525-568
21. Sumi, Y., Etani, T., Fels, S., Simonet, N., Kobayashi, K., Mase, K. (1998). C-MAP: Building a Context-Aware Mobile Assistant for Exhibition Tours. Community Computing and Support Systems, Springer LNCS Vol. 1519, pp 137 - 154
22. Wen, Z., Zhou, M., Aggarwal, V. (2007). Context-aware adaptive information retrieval for investigative tasks. IUI 2008, ACM, pp. 122-131
23. Wirfs-Brock, R., Mc Kean, A. (2003). Object Design: Roles, Responsibilities, and Collaborations, Addison-Wesley